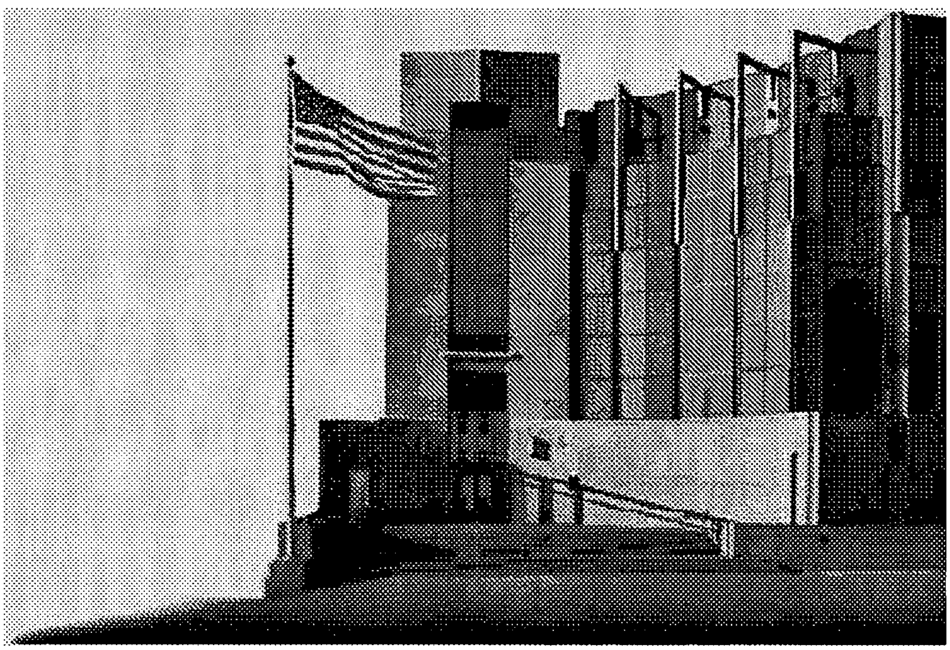
Carnegie Mellon University
Software Engineering Institute

IEEE COMPUTER SOCIETY



Position Papers for the
1997 Information Survivability Workshop - ISW'97
February 12-13, 1997
Catamaran Resort Hotel
San Diego, California
Web Edition
Organized by the Software Engineering Institute, Sponsored by the IEEE Computer Society

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

ISW97 Table of Contents

Welcome

Welcome to the 1997 Information Survivability Workshop (ISW'97). Information survivability (IS) has become a new area of concern for DARPA and others in the research community. IS is more than security, more than safety, and more than fault tolerance. It is a combination of quality attributes that assures that even during a successful attack, the mission of the network, software, or service will continue.

The goal of the Workshop is to clearly define the issues associated with IS, and identify the research areas that are likely to contribute to this area. In the Call for Participation we encouraged participation of people with diverse backgrounds and interests including, but not limited to, researchers, practitioners, research sponsors, and buyers. We are very happy with the enthusiastic response we received and we have assembled a program that reflects the wide diversity of backgrounds and interests in the area of IS. We hope that this meeting is the first step towards the identification of a network of experts that will continue to collaborate and share results to the benefit of the entire community.

We limited the attendance to 50 participants and structured the workshop around two "themes" and each day will be dedicated to one of the themes. To have more time for discussions and exchanges among the participants, we have selected a very small number of speakers. We selected the speakers based on position statements that highlight the range of issues around each of the themes. This is no reflection on the quality of the other statements as we decided that for this first workshop we should explore the breadth of issues rather than focus on specific, narrower topics.

We wish to acknowledge the assistance of Mary-Kate Rada, of the IEEE Computer Society Staff, for her help with advance registrations and local arrangements. Tracey Tamules of the SEI staff assisted with on-site registrations.

- Mario R. Barbacci	- Thomas A. Longstaff	- Howard F. Lipson
- General Chair	- Program Co-Chair	- Program Co-Chair

Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Note:
The frames version of this site has been removed, so as to better conform to the entire CERT site structure. If there are any problems or other comments purely on the web page structure and formatting, please send them to me.
Thanks,
Mark Firth (webmaster-isw97@cert.org)

Performance Signatures: A Mechanism for Intrusion Detection

David L. Oppenheimer and Margaret R. Martonosi
Princeton University
{davido,martonosi}@princeton.edu

A survivable computer system should be able to identify anomalous program behavior, quarantine software components that are misbehaving, and develop and distribute vaccines that eliminate the originally-exploited vulnerability [1]. Focusing primarily on attacks against host systems, we believe that research into how to perform these tasks can benefit significantly from the techniques and tools originally developed for another domain concerned with observing and modifying program behavior: software performance monitoring. To this end, we propose the *performance signature* as a mechanism to detect anomalous program behavior indicative of a hostile attack, and as a metric to guide the automated development of a patch to correct the program flaws exploited in the attack.

1. Performance Monitoring as a Model for Anomaly Detection

The analogy between monitoring performance and detecting anomalous behavior is a natural one. Data commonly collected for performance *monitoring* purposes includes the number of references to particular memory locations, the time spent executing different parts of a program, the frequency and quantity of communication among nodes of a multiprocessor system, and I/O resource usage. Performance *analysis* aggregates this data, extracts observations of interest to programmers, and displays the conclusions in ways that clearly expose the performance impact of those observations. Performance *tuning* may be carried out by hand or by a compiler. In either case the performance tuner "reasons" about the collected performance data, uses it to guide program modifications, and then tests the modifications by re-executing the program, gathering new performance data, and comparing the behavior of the original program to that of the revised one. This monitor-analyze-tune cycle continues until the program's performance becomes acceptable.

Detecting and correcting anomalous program behavior triggered by an attack can be viewed as a similar process. While excessive or inefficient resource usage raises the red flag in performance tuning, it is deviation from "normal" patterns of operation which signals potential system subversion. *Monitoring* for anomalous behavior therefore requires defining the variables that might indicate an attack and continually observing these variables during system operation. The values of these variables during program execution constitute the *performance signature* of the program. *Analyzing* this data entails

issuing a warning when one or more monitored variables enter a "suspicious" range and aggregating anomaly reports from multiple systems. If the anomaly is deemed to reflect an attack, the anomaly reports can be used as the starting point for vaccine development. *Developing a vaccine* involves producing a software patch designed to repel an attack in progress, eradicate the infection, and harden the system against a future attack which exploits the same vulnerability. The success or failure of this patch can be measured by comparing the values of the anomaly-indicating variables in the patched system to those values prior to the patch. An iterative process of developing and testing patches will hopefully converge on one which returns the behavior of the attacked system back to normal.

2. A Proposal for Anomaly Detection and Vulnerability Repair

2.1. Monitoring. Detecting anomalous program and system behavior requires first defining the variables to be monitored. For each variable or group of related variables, one must also define the criteria for suspicious behavior. This might take the form of ranges of values considered to represent an irregularity, or a statistical model that detects meaningful changes in normal patterns of those values. Many variables are possible, but we believe that the statistics typically associated with performance monitoring can be useful metrics for detecting anomalous system behavior. Examples of such variables include time spent in certain program functions, patterns of memory usage, quantity and destinations of network communications, and time spent in code representing specific operating system services. A correctly operating program uses these resources within certain bounds; its usage pattern can be represented as a performance signature. Deviation from this signature may indicate an attack. Monitoring can be performed in a number of ways, with different tradeoffs between statistics detail and runtime overhead. Basic block profiling can gather some execution characteristics with typically less than 50% overhead (*e.g. qpt2 [2]); even full system monitoring (including detailed behavior such as all memory references made by all applications and the operating system) can be performed with slowdowns of roughly 10X (e.g. SimOS [3]).*

2.2. Analysis. The statistics gathered from instrumented executables or full simulation can be processed locally by a trusted system daemon or can be sent to remote machines for analysis. The analyzer compares the program's execution profile to the "normal" performance signature for that program. Sensitive programs could be distributed with an initial performance signature, much as programs today are often distributed with a cryptographic signature of the program's binary executable image. This would allow a program to be monitored immediately, before a local profile history has been accumulated. The definition of a "normal" signature would be refined over time as the program is run. Profile values that fall outside the accepted range or otherwise represent a deviation from expected patterns signal a possible attack. At this stage a misbehaving system would be quarantined by bringing it offline or by limiting its contact with other machines and offloading important functions to other systems.

2.3. Developing a vaccine. The final step uses the collected data to develop a "vaccine" patch for the affected program. Automating patch development from performance signature data is a daunting task. Clearly the space of possible program transformations for correcting security flaws is larger than that for performance tuning. We assume that the system developing the patch (the "security host") has source code for the affected program. In that case the security host can work to identify the specific portions of the program which the attack has targeted and can use the source code to "reason" about the program flaws which may have permitted the attack. If source code is not freely available, the suspicious performance signature can be sent to the program's vendor for analysis. If

the security host also performs program instrumentation or controls the simulation of monitored programs, it can retarget its monitoring efforts with each vaccine distribution, progressively "zeroing in" on the part of the program under attack.

2.4. Examples. This section sketches a few examples detailing how performance signatures can serve as a useful weapon in the security arsenal. First, the performance signature of a user's Web browser would typically indicate network connections established with many different Web servers over time. If the user falls victim to a Web spoofing attack in which an adversary interposes a malicious Web server between the user and the "real" Web [4], the browser's signature will change as all or a disproportionate number of connections are made to the adversary's Web server. As a second example, consider a program which normally uses a cryptographically strong random number generator to produce one-time session keys. If this program is subverted to use a fixed session key known to an adversary, the program's performance signature is very likely to change: the program will no longer execute the function that generates random keys, or the encryption function will stop reading from the memory location where those keys are stored, or the program will overwrite that location with the fixed key, *etc.* An analogous situation exists for a program that normally encrypts its data but is subverted so as to stop encrypting altogether. Third, consider a mail delivery program which normally uses memory-mapped I/O to write to the end of a user's mailbox. If this program starts reading memory regions of the mapped object which it has never accessed before (*e.g.* in an attempt to read, so as to later convey to another user, the contents of a user's mailbox), the memory reference profile of the program will change. Scenarios of arbitrary changes to the behavior of privileged programs are plausible in light of the common buffer bounds-checking problem, which allows a malicious user to overrun an input buffer by feeding overly long input to a vulnerable program, causing the program to execute instructions of that user's choosing.

In general, our system is more likely to detect the introduction of covert timing channels than other proposed anomaly detection systems that observe only macroscopic parameters like system call behavior [5]. For example, consider a text editor subverted to modulate the number of CPU cycles it consumes in order to communicate its buffer contents to another process running on the same machine. A program subverted in this way will spend many of its cycles in a spin-while-using-up-CPU-time loop. Time would not be spent in that loop when the editor is not using the covert channel. An execution profile that monitors the amount of time spent in each program function will identify this behavior; a trace of system calls made by the editor will not. Similarly, detection of the second and third examples in the previous paragraph relies on execution profiling and memory reference observation, and cannot be achieved through system call monitoring alone. Moreover, as user-level library functions replace operating system calls as the preferred way for programs to efficiently use system services, the range of program activities observable via system call tracing will decrease further. This will increase the need to monitor within applications and their libraries.

3. The Threat Model

The process we have described makes several assumptions about the ability of a system to monitor itself and about what happens to a system when it is attacked. We detail these assumptions here.

- We assume a program or system which is being or has been attacked will behave differently with respect to the monitored characteristics than will a program or system which is operating normally. For example, an attack that modifies a program to

behave maliciously is extremely unlikely to do so without *at least temporarily* changing the program's performance signature. We can control the number and types of monitored characteristics and the sampling rate in order to make this assumption true.

- While the first point was concerned with the performance signature of the attacked code, this second point concerns attacks against the monitoring code itself. We assume an attacker cannot modify the instrumentation code, simulator, or profiling daemon in a way that causes the daemon to report "business as usual" during an attack. An attack will either change the performance signature (as in the first point) or will cause the profile information to fail to be reported in a timely manner.
- If profiling data is analyzed remotely, we assume a secure way to move performance signatures from the local profiling process to the security host. This requirement can be satisfied by having that daemon digitally sign the profile reports.
- The vaccine distribution process relies on the ability of infected nodes to receive and incorporate patches. In some cases, such as a denial of service attack which disables a node's network communication abilities, this is an unreasonable assumption. Barring the existence of a tamper-proof patch reception and installation pathway, a response to this sort of attack will simply quarantine the machine.

4. Related Work

Researchers have suggested detecting system intrusion by monitoring for anomalous user behavior [6], program image on disk [7], or program behavior [5,8,9,10]. Because we collect runtime statistics on a per-program basis, the system we propose is similar to the third approach. In particular, [5] establishes a per-program database of short-range correlations in system calls made by a program and signals an anomaly when an executing program's correlations differ from those stored in the database. This improves on some prior approaches, such as that presented in [8], because it is adaptive and requires no *a priori* specification of permitted program actions. But an attacker who knows that this intrusion detection system is in use could specifically engineer his attack to avoid noticeably changing the system call behavior of subverted programs. [9] and [10] also examine only macroscopic program behavior visible to the operating system. As previously mentioned, it is far less likely that an attacker can avoid changing the plethora of variables commonly associated with performance monitoring. Our proposal differs from earlier work in one more key respect. We propose leveraging well-understood and debugged tools that already exist in the performance monitoring community.

5. Research Challenges and Conclusions

This paper has presented a model for adding performance signature information to the arsenal of techniques used to detect system security violations. Key challenges must be overcome in order to maximize the effectiveness of this technique. Although there are a number of possible performance metrics to be measured (and tools to measure them), we must determine a set that detects intrusions well without imposing large software overheads. The set of metrics is likely to be custom-chosen for individual systems and pieces of software. A second key policy issue is developing criteria for distinguishing suspicious performance signatures from those caused by normal system variations. These criteria will depend on desired tradeoffs between security level and tolerance of false alarms. A third observation is also relevant. Security experts recognize that a diversity of program implementations generally aids security, since it is unlikely that a single attack will be effective against all the variations of a program [1]. But these variations mean that each program will display a slightly different performance signature, making out-of-bounds performance behavior harder to pinpoint.

In conclusion, anomaly detection and self-healing are essential parts of a survivable system. How to implement either goal is is very much an open problem. But the logical steps in performing these tasks--identifying undesirable program or system behavior, localizing its cause, and correcting it--greatly resemble the logical steps of performance tuning. We therefore feel that research into building survivable systems should give serious attention to techniques initially developed for software performance analysis. In particular, we suggest performance signatures as an additional mechanism to distinguish normal from abnormal program behavior, and as a metric to guide patch development.

6. References

[1] Howie Shrobe. ARPATech '96 information survivability briefing. *ARPATech '96 Systems and Technology Symposium,* May 1996.

[2] James R. Larus. Efficient program tracing. *IEEE Computer,* vol. 26, no. 5, May 1993, pp. 52-61.

[3] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer simulation: the SimOS approach. *IEEE Parallel and Distributed Technology,* Fall 1995.

[4] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: an internet con game. Princeton University Department of Computer Science Technical Report 540-96, 1996.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, 1996.

[6] Teresa F. Lunt. Detecting intruders in computer systems. *1993 Conference on Auditing and Computer Technology,* 1993.

[7] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, 1994.

[8] Karl Levitt, Calvin Ko, and George Fink. Automated detection of vulnerabilities in privileged programs by execution monitoring. *1994 Computer Security Application Conference*, 1994.

[9] Dorothy E. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, Feb. 1987, pp. 222-232.

[10] Debra Anderson, Teresa F. Lunt, Harold Javitz, Ann Tamaru, and Alfonso Valdes. Detecting unusual program behavior using the statistical component of the Next-Generation Intrusion Detection Expert System (NIDES). Computer Science Laboratory, SRI International, Technical Report SRI-CSL-95-06, 1995.

Back to the Table of Contents

[24]